Distributed Management of Concurrent Web Service Transactions

Mohammad Alrifai, Peter Dolog, Wolf-Tilo Balke and Wolfgang Nejdl

Abstract — Business processes involve dynamic compositions of interleaved tasks. Therefore ensuring reliable transactional processing of Web services is crucial for the success of Web service-based B2B and B2C applications. But the inherent autonomy and heterogeneity of Web services render the applicability of conventional ACID transaction models for Web services far from being straightforward. Current Web service transaction models relax the isolation property and rely on compensation mechanisms to ensure *atomicity* of business transactions in the presence of service failures. However, ensuring *consistency* in the open and dynamic environment of Web services, where interleaving business transactions enter and exit the system independently, remains an open issue. In this paper we address this problem and propose an architecture that supports concurrency control on the Web services level. An extension to the standard framework for Web service transactions is proposed to enable detecting and handling transactional dependencies between concurrent business transactions. We also present an optimistic protocol for concurrency control that can be deployed in a fully distributed fashion within the proposed architecture. We also empirically evaluate the performance of the proposed solutions in terms of throughput and response time.

Index Terms— Software Architectures, Transaction Processing, Concurrency.

1 Introduction

The services computing paradigm and its realization through standardized Web service technologies provide a promising solution for the seamless integration of business applications. Service providers describe the Web interface of their local business processes using WSDL documents and publish them as Web services (e.g. via UDDI repositories), whereas the business logic and local resources management behind the published services remain invisible to the outside world. Web service-based business-to-consumer (B2C) and business-to-business (B2B) applications usually involve invocations of services running on different heterogeneous back-end systems managed by autonomous service providers. BPEL4WS [25] is a workflow-like definition language that can be used to describe complex business processes and orchestrate the involved Web services.

A key requirement for successful Web service-based business applications is to ensure reliable execution of their processes with respect to the partners' transactional requirements [1, 16]. A reliable transaction processing should provide the illusion that each transactional process executes as if no other process were executing concurrently (*serializability*) and as if there were no failures (*recoverability*) [9]. In contrast to traditional ACID transaction models, which assume short lived transactions, transactional Web service-based processes are usually long-running processes (in the order of hours or even days). Therefore, strict isolation requirements to guarantee serializability of distributed transactions have to be relaxed. A bank service provider, for example, would not accept locking its local resources (customer accounts) on behalf of some client application for unbounded time.

Advanced Transaction Models (ATM) have been proposed in the literature to address the new requirements of advanced applications (see [7] for a comprehensive overview). The Open Nested Transaction Model [10] was widely adopted by industry [e.g. 23, 26] and academia [e.g. 9, 16, 6] for Web service transactions. The main feature of this transaction model is the possibility to relax the isolation property by exploiting semantic properties of operations, which allows participants to commit independently (thus, preserving autonomy). The concept of compensation plays a major role in this model to "repair" the semantic effects of already completed activities in the case of failures or transaction abort requests. Many commercial companies nowadays (e.g., Amazon.com) that provide transactions without isolation in their online services, also provide semantic compensation mechanisms (usually in the form of canceling an order within a given time limit). However, in the open and dynamic Web service environment, business transactions enter and exit the system independently. Under isolation relaxation transactional dependencies can emerge among independent business processes, which need to be taken into account when compensation is required in order to avoid inconsistency problems. Such transactional dependencies are currently overlooked in the Web service transaction models and standards.

In our previous work [2, 3] we presented some preliminary ideas towards solving the consistency problems of Web service transactions. In particular, [2] proposed an optimistic variant of the SGT protocol for

- *M. Alrifai, W.-T. Balke and W. Nejdl are with the L3S Research Center, University of Hannover – Appelstr. 9a, D-30167 Hannover, Germany. E-mail: alrifai@ L3S.de, balke@L3S.de and nejdl@L3S.de.*
- *P. Dolog is with the Department of Computer Science, University of Aalborg - Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark. E-mail: dolog@cs.aau.dk.*

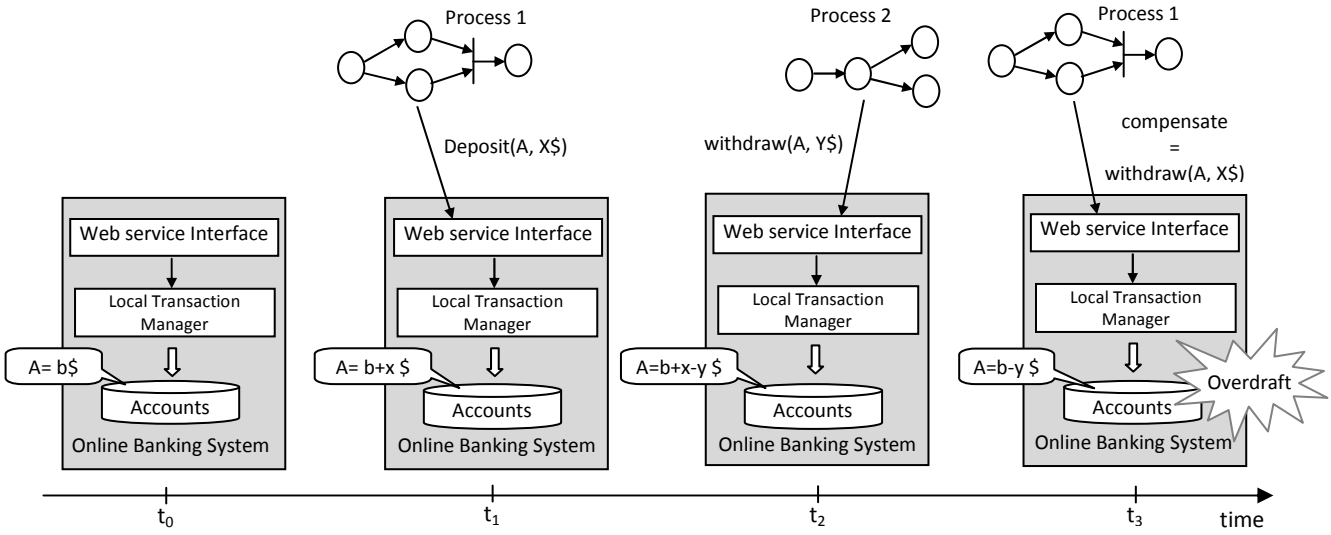

Fig.1 Example of transactional dependency between two processes

concurrency control that can be applied in a fully decentralized manner. The protocol applies a commit-differing policy to ensure the consistency of concurrent executions. An edge chasing algorithm is used to detect potential global waiting cycles spanning several service providers. In [**3**] we targeted the concurrency control problem from another perspective, i.e. from the service composer viewpoint. We proposed a transaction scheduling algorithm that selects the best service providers based on their local scheduling offers. By taking the time requirements of service requesters and providers into account, the algorithm avoids unnecessary waiting times due to transactional dependencies. In this paper we extend our previous work as follows:

- We present a theoretical foundation for the proposed transaction-aware architecture (Section 3).
- We combine pre-scheduling from [3] with the concurrency control from [2] to eliminate the communication costs caused by the edge chasing algorithm (Section 5).
- We study the performance of the proposed solutions by extensive simulations (Section 6).

Besides, after motivating the problems with a concrete scenario (Section 2), we describe implications of our theoretical framework on current industry standards for Web service transaction coordination (Section 4) and compare our approach to related work (Section 7).

2 A Use Case Scenario

The following example demonstrates the problem of maintaining global consistency in the presence of concurrent service invocations and motivates the need for concurrency control at the Web services level. In the example shown in Figure 1 two independent business processes (Process 1 and Process 2) access the Web service of some online banking system concurrently. Both processes update the same account (account A) through two subsequent calls to the deposit and withdraw operations respectively. Assume that the initial balance of account A at time t_0 is b$ and that the bank does not allow overdrafts (i.e. A > 0 must always be true). Process 1 invokes the Web service at time t_1, requesting depositing account A with x$. The balance of account A is now updated to (b + x)$. Due to the isolation relaxation policy, the new balance is immediately made visible to other processes, even before Process 1 has been completed. At time t_2 Process 2 invokes the web service requesting a Withdraw operation of y$ (y > b) from the same account. Accordingly, the balance of account A is updated to (b + x –y)$. Note that without Process 1's deposit operation being successfully executed, the withdrawal operation of Process 2 cannot be accepted by the online banking system as it would lead to an overdraft. Assume now that later, due to the failure of some activities of Process 1, its coordinator decides to cancel the whole process and issues a compensation request of its previous deposit operation. Process 1's compensation request is received by the banking system at time t_3. The compensation is done by a semantically reverse operation (withdraw in this case) on account A with the same amount of x$. However, such operation is not allowed by the banking system as this would lead to an overdraft (b-y < 0).

This scenario points to the following problems:

- Transactional dependencies can occur dynamically between autonomous processes due to concurrent access to transactional web services.
- The transactional processes are usually coordinated by independent coordinators. The dependency conflicts are therefore invisible to the coordinators.
- Locking based solutions would not be acceptable for service providers, as the conflicting processes are often long running processes.

In this paper we will show, how to deal with these problems by a multi level transaction model and a distributed concurrency control mechanism. We also present and compare two distributed solutions for handling global dependency cycles: the edge chasing

approach for detecting global cycles at commit time, and the pre-scheduling approach for avoiding global dependency cycles.

3 Theoretical Foundations for Transaction-aware Architecture

In this section we introduce an architecture that supports Web service concurrency control in a modular way. The architecture distinguishes between service-level and resource-level concurrency control. Figure 2 shows the conceptual view of this multi-layered architecture. Resource-level transactional conflicts (e.g. select/update queries to a DBMS) are managed by the Resource-level Concurrency Control component (e.g., the transaction manager of the DBMS). Transactional dependencies between interleaving processes, caused by service-level semantic conflicts (e.g. deposit/withdraw conflicts), are managed by the Service-level Concurrency Control component. This separation allows supporting any back-end system and concurrency control protocol (e.g. 2PL, multiversion, etc.) [21].

3.1 A Multi-level Transaction Model

The proposed architecture can be elegantly modeled with the multilevel nested transaction model from [18, 19]. This model has a sound theoretical foundation and fits well to multi-layered architectures where each layer has its own level-specific semantics of the set of operations. The model is a special case of the open nested transaction model with the requirement that all leaf nodes in a transaction tree have the same distance to the root. The nodes in a transaction tree correspond to operations at particular levels of abstraction, where the edges represent the implementation of each operation at level L_i by a sequence of operations at the next lower level L_{i-1} (for i=1,…, n in bottom-up order of a n-level system). In the architecture shown in Figure 2 we have a 3-level system (L_1= resource level, L_2=service level and L_3=process level).

To map it onto our scenario, at the process level (L_3) there can be a set of business processes that invoke the online banking service in the context of some business activities. In the service level (L_2) we have the web service interface, which provides an access to the customers' accounts for online banking. For the sake of simplicity, we assume that each activity on the process level is mapped to one web service operation (e.g. a deposit/withdrawal operation). The resource level (L_1) is a level of some database where the customers' accounts are managed. Each web service operation is mapped to a database transaction on the resource level. The consistency of the overall system can only be guaranteed, if the produced schedule at each level is guaranteed to be serializable (i.e. equivalent to some serial execution of the involved transactions) [18]. We restrict our focus on process-level transactions with Web service level operations as elementary operations of these transactions. The correctness of access to low level resources is then left to the resource-level transaction manager (e.g., a DBMS).

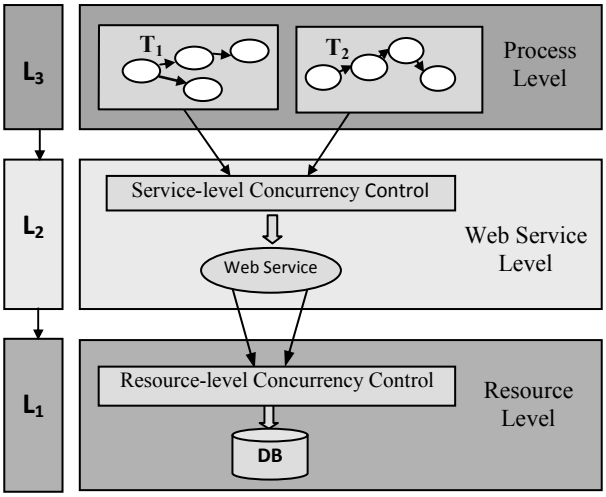

Figure 2. The Multi-layered Architecture

3.2 Transactional Dependencies

There is a dependency relation between two process-level transactions T_1 and T_2 if the outcome of T_2 is influenced by the outcome of T_1. We formally define this relation as follows:

Definition 1: Transactional Dependency
There is a *transactional dependency* between two transactions T_1 and T_2, from level L_3 if there are two activities (service operations) $a_1 \in T_1$ and $a_2 \in T_2$ from level L_2 such that:

- the failure of a_1 causes the failure of a_2 or
- the success of a_1 causes the failure of a_2, and

We refer to T_1 as the *dominant* transaction and to T_1 as the *dependent* transaction. In our online banking scenario, Process 2 is dependent on Process 1. The transactional dependency relation is analogous to the semantic conflict relation in database transactions terminology [19]. Ensuring consistency of business transactions requires tracking these dependencies and handling them appropriately. This can be achieved by building and maintaining the so-called *dependency graph* (analogous to the serialization graph in databases) and ensuring it contains no cycles.

Dependency Graphs

A dependency graph is a directed graph where the nodes represent transactions and the edges represent transactional dependencies between them. Each edge points from the dependent transaction to the dominant one. Dependency graphs are updated whenever a new transaction enters or leaves the system. A transaction with no outgoing edges (i.e., it has no dominants) is said to have an *exclusive lock* on the shared resources. All its dependent transactions are said to have *shared locks* on

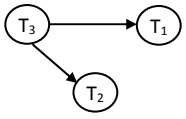

Figure 3. Example of a Dependency Graph

these resources. Figure 3 depicts an example of a dependency graph composed of three active transactions. In this example, transactions T_1 and T_2 have exclusive locks, whereas T_3 has shared locks with T_1 and T_2. The direction of the edges indicates that the outcome of T_3 *depends on* the outcome of both T_1 and T_2. Therefore, a concurrency control mechanism is required to detect such dependencies and ensure that T_3 does not leave the system before T_1 and T_2 successfully terminate.

3.4 Global Consistency through Local Guarantees

In the distributed and open environment of Web services, independent business process transactions can co-exist without knowing about each other. Under the absence of a global transaction manager, maintaining the global dependency graph is therefore not feasible, and a distributed management of the transactional dependencies is required. To this end, we refer to the results of extensive research in the databases field (see [20] for a comprehensive overview). It has been shown that global consistency can be achieved through strong local guarantees: i.e., by ensuring that each local transaction schedule satisfies either the Rigorousness (RG) or the Commit-Order preservation (CO) criteria [20]. The Rigorousness criterion requires isolating the intermediate results of active transactions until the termination (commit or abort) of all preceding transactions. As a result of this strict policy, the concurrency level of the system decreases significantly, which negatively influences the overall performance. A de facto standard implementation of a rigorous concurrency control strategy is the well-known (strict) 2PL protocol.

On the other hand, the Commit-Order preservation criterion permits accessing data processed by active transactions under the constraint that the commit order of any two semantically conflicting transactions must preserve their execution order. This practically means delaying the commit of transactions in conflict till after the commitment of their preceding transactions. Commit-order preserving schedules therefore allow higher level of concurrency than rigorous schedules, which leads to better performance and higher overall throughput. However, this gain in performance does not come without costs. Commit-order preserving schedules run under the risk that distributed transactions might get blocked by other relatively longer transactions at commit time which can lead to unacceptable waiting times. In this paper, we adopt the Commit-order preserving policy for concurrency control and propose solutions to cope with the aforementioned limitations.

4 Extending Web Service Transaction Framework with Multi-Level Transaction Model

OASIS has approved the WS-Coordination [24], WS-AtomicTransaction [22] and WS-BusinessActivity [23] specifications (from IBM, Microsoft, BEA and others) as the de facto standards for Web service transactions. In the following we give an overview on the different components of the WS-Transaction Framework as described by the OASIS specifications. We then propose an extension to this framework for supporting concurrency control.

4.1 Components of Current WS-Transaction Framework

WS-Coordination [24] defines a framework that provides a coordination context for all loosely coupled partners in a distributed application. Within this context application-dependent protocol, messages can be exchanged among the partners. The framework defines two key concepts (see Figure 4):

1. The *coordinator* is responsible for creating the context and coordinating the different partners according to the applied protocol. The coordinator role can be taken by the initiator of a distributed application or by a (trusted) third party.

2. The *participant* is an entity that resides on the Web service provider side and represents an instance of the Web service that has been invoked within the distributed application. This entity is responsible for communicating with the coordinator according to the applied protocol on behalf of the Web service.

Transaction protocol's massages can be exchanged and coordinated within this framework. There are currently two transactional coordination types: WS-AtomicTransaction and WS-BusinessActivity.

WS-AtomicTransaction [22] supports traditional ACID transactions and is intended for short-duration interactions among trusted partners. Under the scope of an AtomicTransaction, the coordinator directs all participants to either all commit or all cancel using the well known 2PC protocol.

WS-BusinessActivity [23] on the other hand is intended for long-duration and ACID-relaxed transactions among loosely-coupled systems where locking resources is impractical or not desirable. The WS-BusinessActivity coordination type is based on the open nested transaction model [10, 15]. Transactions in this model can form a tree (of arbitrary height) of "sub-transactions". The sub-transactions may commit independently of each other without having to wait for the root transaction to commit. In case of a sub-transaction failure, the client driving this business process may decide whether the overall

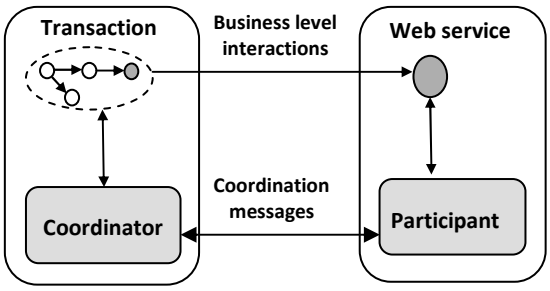

Figure 4. Web Service Transaction

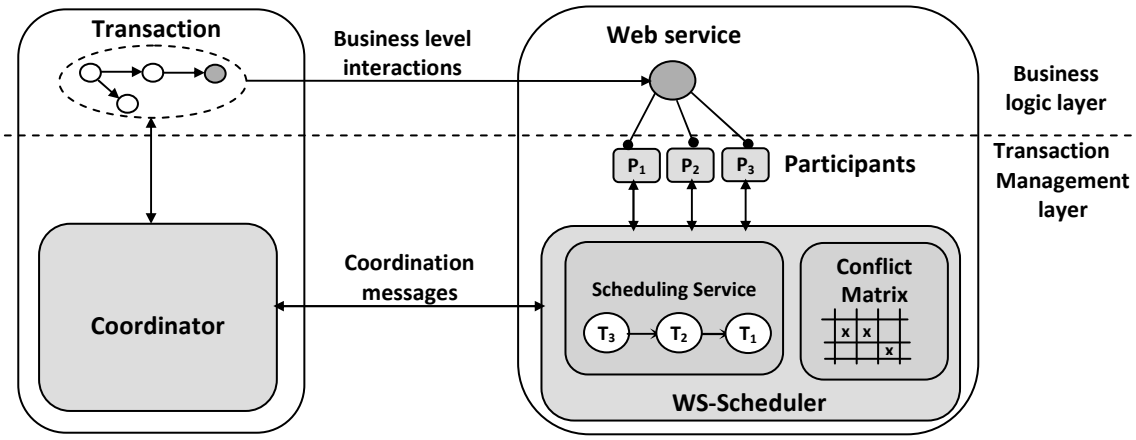

Figure 5. Extended Web Service Transaction Framework

transaction should abort or simply ignore the failed sub-transaction. The open nested transaction model, and hence, the WS-BusinessActivity coordination type also relaxes the isolation property. It permits disclosing intermediate results by autonomous participants instead of locking local resources until the end of the (global) transaction. In the case of transaction abort, the effects of already committed subtransactions are undone by means of compensating subtransactions. However, the assumption that all service operations can always be compensated is not realistic. When the number of transactions having access to intermediate results increases, the compensation of some operations becomes either too expensive or even impossible. This raises the need to a concurrency control mechanism for Web service transactions.

Supporting concurrency control for the WS-BusinessActivity is challenging for the following reasons. First, business activities are usually long-running, which yields locking-based solutions (e.g. 2PL protocol) unacceptable. Second, participants of a business activity are loosely-coupled and highly autonomous, which makes global scheduling based solutions (e.g. global serialization graph testing protocol) inapplicable.

4.2 WS-Scheduler for Concurrency Control

We extend the current standard Web service transactions framework by introducing the *WS-Scheduler* to implement the service-level concurrency control. The WS-Scheduler resides on the Web service provider's side and is responsible for managing concurrent instances of the WS-Coordination (and WS-BusinessActivity) protocol. Figure 5 shows how the WS-Scheduler is integrated into the standard framework. It maintains a list of active participants, and the transaction contexts they are involved in. This can be easily implemented as part of the invocation mechanism. As every service request within a global transaction is associated with the coordination context according to the WS-Coordination standard [24], the context is extracted from the received message and a new participant is created. The control is then transferred to the Scheduling Service of the WS-Scheduler, which (on behalf of the created participant) registers itself as a participant of the given context. This is done by invoking the registration service at the given coordinator address. The Scheduling Service then checks if there are transactional dependencies between the requested service operation and previously executed operations of active transactions. The WS-Scheduler maintains a dependency graph and decides (based on the deployed concurrency control mechanism) when transactions are allowed to commit their activities and leave the system. This ensures that commit and compensation requests are handled consistently. In the following sections we describe in detail how potential transactional conflicts are detected and handled.

In the extended framework (Figure 5) all coordination (response) messages from the coordinator (participant) are received and processed by the WS-Scheduler before forwarding them to the participant (coordinator). For example, when a commit message is received from the coordinator, the WS-Scheduler has to check first whether this transaction is allowed to commit before forwarding this message to the respective participant. Consequently, all the states of the web services in the different contexts (e.g. completing, compensating, aborted etc) as defined by the WS-BusinessActivity specification are kept by the WS-Scheduler. In Figure 6 we refer back to the use case scenario of Section 2. The two processes Process 1 and Process 2 access the online banking service within two independent contexts. The two contexts are coordinated by two autonomous coordinators WS-Coordinator 1 and WS-Coordinator 2 and represented locally by two participants P_1 and P_2 respectively (Figure 6.a). Potential transactional dependencies between the two contexts cannot be detected as they are not visible to the two coordinators. With the deployment of the WS-Scheduler as shown in Figure 6.b, such dependencies can be easily detected and appropriately handled.

4.3 Detection of Transactional Dependencies

A conflict matrix is built at design time by the service provider to assist the WS-Scheduler in detecting potential transactional dependencies at run time. The conflict matrix is an NxN matrix, where N is the total number of operations that can be invoked via web service calls. The

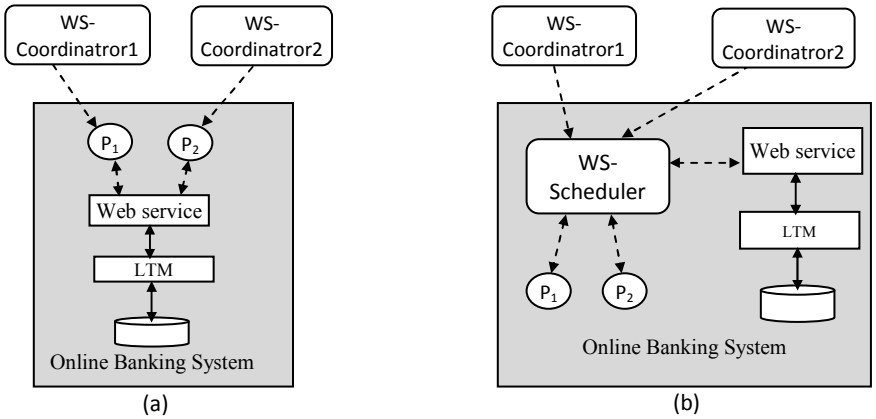

Figure 6. Using the Web Service Transaction Framework with the Online Banking Service:
a) The Web Service is concurrently involved in two different transactional contexts
b) WS-Scheduler managing the two contexts

conflicts can be defined based on the semantics of these operations (i.e. based on their behavior and effects) to reflect their execution commutativity relations [19]. Two operations do semantically conflict if they do not commute, i.e. if changing the order of their execution results in different final state.

Consider our Banking Web service example, and assume that the service has three operations: deposit(A, x$), withdraw(A, x$) and getBalance(A). A deposit (or withdraw) operation performs a credit (or debit) action on the requested account A with the specified amount of money (i.e. x$). A getBalance operation returns the current balance of the specified account and writes the returned value into a log record. According to this functional description a withdraw operation would semantically conflict with a deposit operation if it was called while the transaction that invoked the deposit operation is still active (i.e. neither committed nor aborted). On the other hand, invoking the deposit operation after a withdraw operation can be tolerated. The operation getBalance does not conflict with any other operation in this example.

However, this definition of semantic conflicts has its limitation: decisions about conflicts are made independently on the accessed resources and their status at run time. Therefore, we extend the commutativity-based conflict definition to capture the dynamic nature of semantic conflict relations. We use conflict predicates, which can be defined by the service provider at design time and evaluated at request time to detect any transactional conflicts. The conflict predicate takes input parameters (e.g. account number) and the current status of the targeted resources (i.e. current balance) as parameters and returns either a TRUE (i.e. conflicting) or FALSE (i.e. not conflicting). For example, in our Banking Web Service example, a conflict predicate for the deposit(A_1, x_1)/withdraw(A_2, x_2) operations would check if the two operations access the same bank account (i.e. if $A_1 == A_2$), and also compare the requested amount of money (x_2) with the last committed balance ($b\text{-}x_1$). If the requested amount of money is greater than the current balance a conflict is detected and the predicate evaluates to TRUE and it evaluates to FALSE otherwise.

The conflict predicate for state-independently non-conflicting operations always returns a TRUE, while the conflict predicate for state-independently conflicting operations always returns a FALSE. Table 1 gives an example of a conflict matrix for the Banking Web Service including the conflict predicates. In this paper we assume that semantic conflict matrices are built and updated by the service provider and made accessible to the WS-Scheduler.

Table1: Conflict Matrix of the Banking Web Service

Last executed / New request	Withdraw (A_1, x_1)	Deposit (A_1, x_1)	getBalance (A_1)
Withdraw(A_2, x_2)	FALSE	$A_1 == A_1$ AND $x_2 > b - x_1$	FALSE
deposit(A_2, x_2)	FALSE	FALSE	FALSE
getBalance(A_2)	FALSE	FALSE	FALSE

5 Distributed Concurrency Control For Web Services

Extending the standard Web service transactions framework with WS-Schedulers enables the detection of transactional dependencies between concurrent processes. However, once detected, these dependencies need to be handled appropriately to ensure consistent outcome of the dependent processes. In this section we present a concurrency control mechanism for this purpose. We introduce a concurrency control protocol, which is a distributed variant of the conventional Serialization Graph Testing protocol [21] and implements the Commit-Order preservation policy [20]. We also present a distributed solution for handling global dependency cycles.

5.1 Optimistic Distributed SGT Protocol

The proposed protocol is a distributed variant of the original Serialization Graph Testing protocol (SGT) [21]. The SGT protocol maintains a graph representation of the

transactional conflicts among active transactions, called serialization graph (i.e. dependency graph in our case). The global *serializability* of the concurrent transactions is guaranteed by ensuring that the graph always remains acyclic. We implement a distributed variant of the SGT protocol, in which every WS-Scheduler maintains a local view of the global dependency graph. The local sub-graphs capture dependency relations among transactions that have active invocations to local services. Each WS-Scheduler ensures that its local dependency sub-graph remains acyclic. WS-Schedulers applying the Commit-Order preserving policy to control the commit order of concurrent transactions. As discussed earlier in section 3.4, this is an optimistic concurrency control policy, in which concurrent access to local services is accepted immediately, while a consistency check is made at the commit time. The consistency of transactions' outcome is ensured by the WS-Schedulers by applying the following two rules:

1. A transaction is only allowed to commit after all its dominant transactions have committed.
2. When a transaction aborts and/or compensates its local activities, the local activities of all its dependent transactions are compensated automatically.

As a consequent of the first rule, any commit request issued by a dependent transaction is delayed until all its dominants have committed. Therefore, we add a new state, i.e. the *waiting* state, to the defined states of the WS-BusinessActivity specification [23]. An extended version of the abstract state diagram of the WS-BusinessActivity's BusinessActivityWithCoordinatorCompletion protocol including the new *waiting* state is shown in Figure 7**.** In addition to the defined message types, we add the message WAIT in our protocol to inform the coordinators that their commit request hast to be delayed due to consistency reasons. The WS-Scheduler keeps track of the current state of all concurrent participants and their transactional contexts. As soon as all dominants of a waiting transaction commit, the delayed commit request is forwarded to the web service and the transaction's coordinator is informed by a COMPLETED message.

The example shown in Figure 8 shows how the WS-Scheduler controls the commitment of transactions based on the local dependency graph. The commit requests (i.e. COMPLETE messages) from transaction coordinators are received by the WS-Scheduler on behalf of the participants. The WS-Scheduler then checks his local

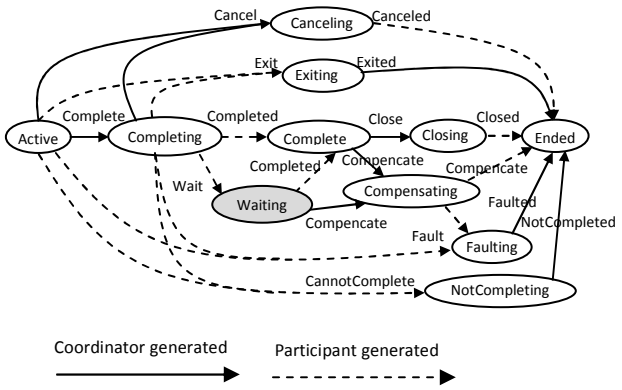

Figure 7. Abstract State Diagram of `BusinessAgreementWithCoordinatorCompletion` with the new waiting state

dependency graph to determine whether this transaction has outgoing edges. In Figure 8.a the WS-Scheduler finds some outgoing edges of the transaction node (i.e. the transaction is holding a shared lock) and decides to delay the commitment of this transaction until all its dominants terminate. The COMPLETE message is not forwarded to the participant and a WAIT message is sent to the coordinator. This is important to avoid conflicting situations like the one given in the Online Banking Service scenario from Section 2. The dependency relation between the deposit and withdrawal operations can be defined in the conflict matrix at design time. Using this information, the WS-Scheduler can detect at run time the dependency between Process 1 and Process 2. Accordingly, an edge from Process 2 to Process 1 is added to the local SGT sub-graph. The WS-Scheduler delays the commitment of Process 2 until Process 1 terminates. This ensures that Process 1 can compensate its deposit operation safely when required. The compensation of Process 1's deposit operations triggers the compensation of Process 2's withdrawal operation automatically, in order to preserve the overall consistency of the system. Figure 8.b depicts another situation, in which the WS-Scheduler finds that the transaction node in the sub-graph has no outgoing edges (i.e. holding an exclusive lock). The commit request is accepted and forwarded to the corresponding participant immediately. Upon receiving the completed message from the participant, the WS-Scheduler removes the node from its graph and forwards the COMPLETED message to the coordinator.

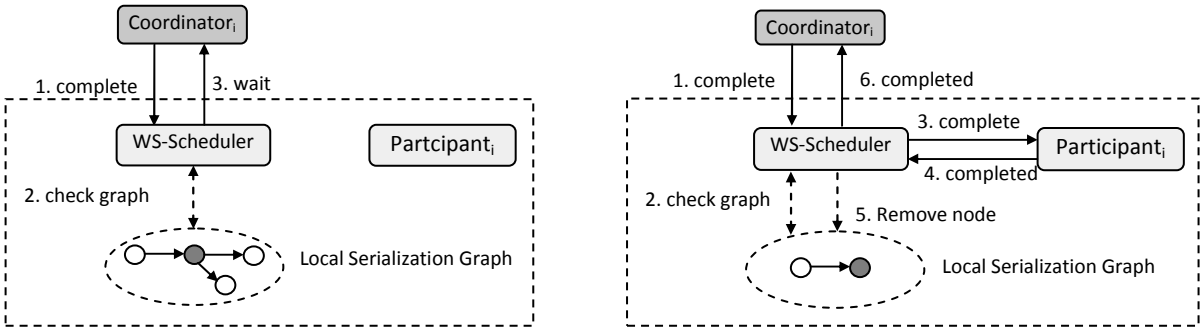

Figure 8. WS-Scheduler decides upon commit requests: a) Delayed Commit b) Immediate Commit

WS-Scheduler Protocol

The WS-Scheduler protocol is presented in Algorithm 1. The WS-Scheduler maintains the SGT local sub-graph, a list of local web services and a list of active participants and their transactional contexts including their current states. The WS-Scheduler takes the Conflict Matrix as an input and uses this matrix for detecting potential dependencies at run time. Lines 3-16 in Algorithm 1 describe how the WS-Scheduler updates its SGT sub-graph upon receiving new service invocation requests. A new node is added to the graph if the invoking transaction (i.e. process) has not yet been added to the graph. Using the Conflict Matrix, the WS-Scheduler detects potential dependencies and adds a new edge from the invoking transaction's node to every conflicting node. The edges indicate that the new transaction is not allowed to commit before all its dominant transactions terminate. The WS-Scheduler then checks if the local dependency graph remains acyclic after adding the new edges. If a cycle is found in the graph, the new service invocation request is rejected and the new added node and all its outgoing edges are removed from the graph.

Lines 18-30 describe how the WS-Scheduler responds to coordination messages from transaction coordinators. When a COMPLETE message is received (i.e. a commit request) the WS-Scheduler forwards the message to the corresponding participant only if the transaction does not have any dominants (i.e. its node does not have any outgoing edges).

Upon receiving the COMPENSATE message from a transaction coordinator the WS-Scheduler triggers the compensation of all its dependent transactions before forwarding the message to the corresponding participant.

Lines 32-40 describe how the WS-Scheduler responds to internal messages from the participants. All messages are forwarded immediately to the corresponding transaction coordinator. In addition, after forwarding a CANCELED or CLOSED or COMPENSATED message, the WS-Scheduler removes the transaction node as well as the calling participant from the dependency graph and the participants list respectively as this indicates that the transaction has terminated.

Algorithm 1: WS-Scheduler Protocol

```
Input:
SG = {} (local serialization graph)
S  = {s1,…,sn} (list of local services)
CM = Conflict Matrix (n x n)
P  = {} (list of active participants)
Start:
1:   while(true) { wait for next event }
2:     // new request for service sj from Transaction Ti
3:     if ( request_received(Ti, sj)) then
4:         if (!SG.contains(Ti)) then
5:             SG.add_node(Ti)
6:             Pi = createParticipant(Ti); P += Pi
7:             SG.set_status(Ti, ACTIVE)
8:         endif
9:         D = get_dominants(CM, SG, sj)
10:        foreach Tk ϵ D → SG.addLink (Ti,Tk)
11:        if (SG.isCyclic) then
12:            reject_request(Ti, sj)
13:            SG.remove(Ti);   P-= Pi
14:            send_message(Ti, CANNOTCOMPLETE)
15:        else execute(sj)
16:    endif
17:    // message m received from coordinator of Ti
18:    if (external_message_received(m, Ti)) then
19:      switch m
20:          case COMPLETE:
21:              if (!SG.has_dominant(Ti)) then
22:                  forward(m, Pi)
23:              else send_message(Ti, WAIT)
24:          case COMPENSATE:
25:              compensate_dependentsOf(Ti)
26:              forward (m, Pi)
27:          case CANCEL or CLOSE:
28:              forward (m, Pi)
29:      endswitch
30:    endif
31:    // message m received from participant Pi
32:    if (internal_message_received(m, Pi)) then
33:      switch m do
34:          case COMPLETED:
35:              forward(m, Ti)
36:          case CANCELED or CLOSED or COMPENSATED:
37:              forward (m, Ti)
38:              SG.remove(Ti); P-= Pi
39:      endswitch
40:    endif
41:  endwhile
```

5.2. Handling Global Waiting Cycles

By preserving the commit-order of transactions, WS-Schedulers can guarantee consistency of their accessed data. However, the distributed implementation of the commit differing policy has a side effect, namely global waiting cycles. These cycles occur as a result of dependency cycles that are neither visible to WS-Schedulers, nor to WS-Coordinators. Figure 9.a depicts an example of such dependency cycles. In the worst case, such cycles can lead to having the transactions waiting for ever.

Definition 2: Waiting Cycle

A waiting cycle is a dependency cycle involving k transactions $T_1, \ldots, T_k$, $k>1$, such that:

1) T_i depends on T_{i+1} for $1 \leq i \leq k$-1
2) T_k depends on T_1
3) T_i is ready to commit, $1 \leq i \leq k$

In [2] we introduced an edge chasing [12] based solution for handling global waiting cycles in a fully distributed manner. However, the communication cost of this approach in terms of the number of exchanged messages for token propagation makes this approach less effective for heavily connected dependency graphs. In this paper, we show how we can eliminate the communication cost of handling waiting cycles by using the pre-scheduling approach instead of the edge chasing approach. To be self-contained, we first present the edge chasing approach, before introducing the pre-scheduling approach.

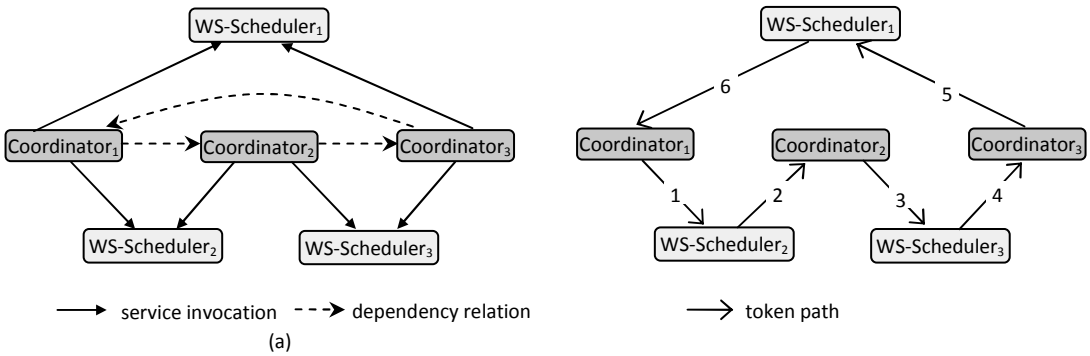

Figure 9. a) Example of a global dependency cycle b) Edge chasing-based cycle detection

5.2.1 The Edge Chasing Approach

According to Definition 2, we consider only dependency cycles in which all involved transactions have already reached the ready-to-commit state. Unlike the proposed cycle detection algorithm in [17], we apply this algorithm in a way that avoids direct communication between independent transactions (Figure 9.b). The cycle detection process can be started by a WS-Coordinator upon the receipt of a WAIT message. The WS-Coordinator creates a unique token (e.g. using the transaction id and client IP address). We call this token a *WaitingCycleCheck*. The token is then sent to the WS-Scheduler that sent the WAIT message. The WS-Scheduler forwards the token to the sender's dominants (according to its local dependency graph). Each of the receiving WS-Coordinators in turn checks the status of its web services and responds as follows. If the WS-Coordinator has no services in the *waiting* state, it replies by sending a *NoWaitingCycle*. Otherwise, the WS-Coordinator propagates the *WaitingCycleCheck* token to the (WS-Schedulers of the) *waiting* web services. This policy ensures that the token is only propagated when all involved transactions are in a waiting state and ready to commit. This is an important condition that adheres to Definition 2 and is useful for the cycle resolution as we will see later. As a result of the cycle detection process, the initiator of this check either receives a *NoWaitingCycle* token or its own *WaitingCycleCheck* token. While the former case indicates that some of the dominant transactions are still busy, which means that the WS-Coordinator has to wait, the latter case indicates the existence of a waiting cycle, which needs to be resolved. Conventionally, waiting cycles are resolved by means of either a complete or partial roll back of (some of) the involved transactions [12]. A victim selection policy is usually applied to select the transactions to be restarted. Note that in the context of Web Services-based business transactions restarting already completed transactions can reduce the performance dramatically. Therefore, we use a forward cycle resolution policy instead. Based on the strong condition that WS-Coordinators forward the *WaitingCycleCheck* token only if they are ready to commit, we allow WS-Coordinators to commit their activities once a waiting cycle is detected, as long as all involved transactions are ready to commit. The readiness to commit is implicitly confirmed by the transaction coordinators when forwarding the *WaitingCycleCheck* token instead of responding with a *NoWaitingCycle* token. As soon as a WS-Coordinator receives his own *WaitingCycleCheck* token, it knows that it is involved in a dependency cycle and that all involved transactions are ready to commit. By committing and closing own activities, the dependency cycle is resolved and other transactions can safely commit as well.

5.2.2 The Pre-scheduling Approach

In contrast to the edge chasing approach, pre-scheduling of transactions solve the waiting cycle's problem without high communication costs. This is useful for environments where the probability of getting into transactional conflicts is very high. For example, when some web services are heavily used by many concurrent business transactions. In such case, using the edge chasing approach would raise a high communication cost for detecting and resolving global dependency cycles. The basic idea of the pre-scheduling approach is to impose some time constraints on using Web services and communicate this information with service requesters. By checking the time constraints of all Web services in a transaction, WS-Coordinators detect timing conflicts that can lead to blocking the transaction at commit time and handle them appropriately prior the actual execution of the transaction. Recall the concept of exclusive and shared locks from Section 3.3. A transaction that does not depend on the outcome of any other transaction is said to have an exclusive lock. This means that this transaction can commit its local changes immediately without any delay. A transaction holding a shared lock, on the other hand, is not allowed to commit until all its dominant transactions terminate and its shared lock is upgraded to an exclusive lock. In the pre-scheduling approach, WS-Schedulers impose time constraints on holding exclusive locks. Any timing conflict between the participating Web services can lead to blocking the transaction at commit time.
In the following we give a formal definition of a transaction schedule.

Definition 2: (Transaction Schedule)
Let T be a business transaction composed of n Web services $WS_1, WS_2, .., WS_n$. Let L_i be the expected time for acquiring the exclusive lock of WS_i and R_i be the deadline for releasing the exclusive lock. A transaction schedule S of T is a schedule in which L_i and R_i are defined for all WS_i in T, $1 \le i \le n$.

Both the expected time for acquiring an exclusive lock of a Web service and the deadline for releasing the lock are specified by the corresponding WS-Scheduler. The WS-Schedulers specify these time constraints based on the concurrency policy of the service provider, the current

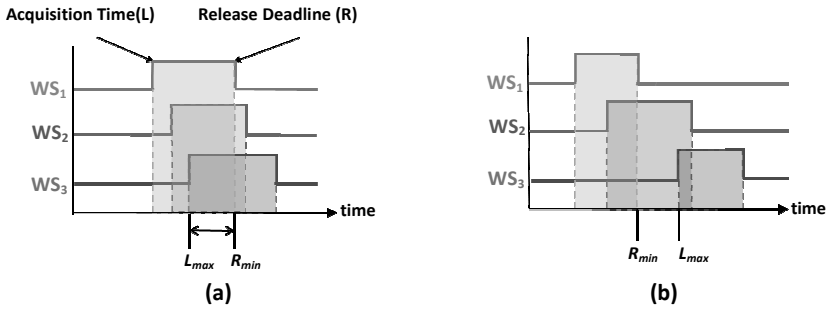

Figure 10. Two schedule examples for a transaction with three Web services:
a) conflict-free schedule b) conflict between WS1 and WS3

status of the local SGT sub-graph and statistical information about the expected service execution duration (i.e. mean and standard deviation values). WS-Schedulers provide the necessary interface for WS-Coordinators (e.g. via a Web service interface) to inquiry about these time constraints when required. In the following we give a formal definition of blocking and nonblocking schedules.

Definition 3: (Blocking vs. Nonblocking Schedule)
Let T={ WS_1, WS_2, …, WS_n} be a business transaction. Let L_{max} = max(L_i), $1 \le i \le n$, be the latest exclusive lock acquisition time and R_{min} = min(R_j), $1 \le j \le n$, be the earliest deadline for releasing an exclusive lock in a schedule S of T. Schedule S is nonblocking iff: $R_{min} > L_{max}$ and is blocking otherwise.

In other words, a transaction schedule is blocking if the acquisition of the required exclusive locks cannot be synchronized, i.e. not all needed exclusive locks can be acquired before the earliest release deadline. The synchronization of the exclusive locks acquisition ensures that all web services are able to commit at the same time. Consequently, the invoking business transaction is not going to be blocked (i.e. delayed) by any of the participating WS-Schedulers at commit time. We explain this further in the following example. Figure 10 illustrates two example schedules for a transaction with three Web services. The schedule in Figure 10.a is an example of a nonblocking schedule. We see that the exclusive lock holding spans (i.e. time span between lock acquisition L and lock release R) of all web services are overlapping, i.e. locks can be acquired (and released) before the earliest provider's deadline R_{min} is expired. Recall that these lock holding spans are given by the participating WS-Schedulers based on the current status of their local dependency graphs. Hence, the overlap between the exclusive lock holding spans implies that the transaction's node in all distributed dependency sub-graphs has no outgoing edges at the period between L_{max} and R_{min}. The WS-Coordinator can safely commit and close its transaction at all sites within this period without any delay. This also implies that the transaction is not involved in any global dependency cycle. The schedule in Figures 10.b, however, has a conflict between the lock holding spans of WS_1 and WS_3 as the activities of WS_3 cannot be committed before L_{max}, which is later than the deadline R_{min} imposed by WS_1.
When a blocking schedule is detected, WS-Coordinators have to handle this by re-scheduling the execution of the transaction at a later point of time that does not lead to the same conflict. The proposed transaction pre-scheduling protocol is presented in Algorithm 2.

The algorithm is executed by the WS-Coordinator prior the actual execution of the business transaction. First, the WS-Coordinator inquiries about the timing constraints for acquiring and releasing the exclusive locks of each of the involved services (Lines 1-5). Each WS-Scheduler checks its local dependency graph and makes an offer for the WS-Coordinator based on their current load at the moment of the request. The WS-Coordinator then checks these constraints to detect any timing conflict (Lines 6-14). There is a conflict if the lock holding spans of all the involved services do not overlap. The WS-Schedulers verifies this by comparing L_{max} and R_{min}, i.e. the latest acquisition time and the earliest release deadline respectively. There is an overlap between all lock holding spans if $L_{max} < R_{min}$, i.e. all locks can be acquired, before the earliest deadline expires. Otherwise, a conflict is detected and the WS-Coordinator repeats the previous steps in randomly set time intervals until a conflict-free

Algorithm 2: Transaction Scheduling Protocol

Input:
T = {WS_1,…,WS_n}, a transaction composing n Web services
Start:
1: **foreach** WS_i in T
2: Let $Scheduler_i$ = WS-Scheduler of WS_i
3: Set L_i = $Scheduler_i$.getLockAcquisitionTime(WS_i)
4: Set R_i = $Scheduler_i$.getLockReleaseDeadline (WS_i)
5: **endforeach**
6: Set L_{max} = max(L_i) , $\forall WS_i \in T$
7: Set R_{min} = min(R_i) , $\forall WS_i \in T$
9: **if** ($L_{max} > R_{min}$) **then**
12: **wait** for a random amount of time
13: **goto** step 1
14: **endIf**
15: **foreach** WS_i in T
16: Set L_i = L_{max}
17: Set R_i = R_{min}
18: **endforeach**

(i.e. nonblocking) schedule is found. Once a conflict-free schedule is found, the WS-Coordinator uses the overlapping interval L_{max} to R_{min} for all web services in the transaction in order to synchronize their commit (Lines 15-18). The WS-Coordinator communicates these values establishes a service Level Agreement (SLA) with all involved WS-Schedulers.

6 Experimental Evaluation

The purpose of this evaluation is to study the performance of our distributed and optimistic variant of the SGT protocol in comparison with the conventional distributed Two Phase Locking protocol (2PL). We experimented with both global cycle handling methods: the edge chasing method and the pre-scheduling method. The performance is measured in terms of average response time (as perceived by the business transaction's initiator) and overall throughput of the system (i.e. number of terminated transactions per second). Our hypothesis that we want to validate, is that our solution outperforms the distributed 2PL in terms of both criteria.

For the purpose of this evaluation, we implemented a prototype for the WS-Coordination and the BusinessActivityWithCoordinatorCompletion protocol according to the BusinessActivity specifications [23]. On the service provider's side, we implemented the WS-Scheduler component and extended the participant's functions to be able to communicate with it. On the client-side, we extended the coordinator's functions to be able to support the cycle detection service as well as the pre-scheduling algorithm.

6.1 Experiment Settings

For experimental evaluation purposes we simulated the environment of concurrently running Web service-based processes. In each experiment we ran a number of concurrent transactions each of them is assigned to a coordinator. Every transaction is composed of a (randomly set) number of tasks; each of them can be accomplished by one of several alternative Web services from different providers. Every call to a service starts a new thread, which performs some transactional operations (read/write) on some local resources (text files). To simulate variant execution lengths of the Web services, we delay the return of the results by a randomly set amount of time following a Pareto distribution. Table 2 summarizes the different parameters of the simulation setup in our experiments.

All experiments were conducted on a machine with a 2GHz Genuine Intel CPU, T2500 processor and 2GB RAM equipped with Microsoft Windows XP Professional Version 2002. The JVM used is J2SE 1.5.

6.2 Response Time vs. Concurrency Level

In this experiment we measure the average response time of the concurrency control methods under different concurrency levels. We execute the transactions in several runs with a different number of alternative web services in each run. By varying the number of alternative web services (from 200 to 40), the probability that transactions invoke the same web service increases, hence, the probability that transactional dependencies among them occur also increases.

In Figure 11, we compare the average response time of the distributed Two Phase Locking protocol (2PL) and our distributed SGT protocol (DSGT). We measure the response time of our protocol twice: once using edge chasing (DSGT_EdgeChasing) and another time using pre-scheduling (DSGT_PreScheduling) for handling global dependency cycles. The measured response time of 2PL is the required time for acquiring all the requested locks. The response time of DSGT_EdgeChasing involves waiting times that are caused by the commit differing policy and the response time of DSGT_PreScheduling is the required time for finding a nonblocking schedule. The results shown in Figure 11 indicate that the average response time of all methods increases as the concurrency level increases (as the number of alternative services decreases). However, the response time of DSGT in both cases increases much slower than the response time of 2PL, which indicates that DSGT is much more efficient and scalable. We also notice that the response time of DSGT_PreScheduling is in average lower than the response time of DSGT_EdgeChasing. This is because in the DSGT_PreScheduling transactions avoid unnecessary waiting times by ensuring that the transaction schedule is conflict-free prior the actual execution. In DSGT_EdgeChasing on the other hand, transactions start execution immediately and delay the dependency check till the commit time. This quite often leads some transactions being blocked by other dominant ones due to the Commit-order preserving policy.

Table 2. Simulation setup

num. of concurrent transaction	100
num. of Web services per transaction	5 to 30
num. of providers per service	40 to 200
Service execution length	5 to 30 sec
Distribution of service execution length	Pareto
Shape parameter of Pareto Distribution	$\alpha = 3$
Scale parameter of Pareto Distribution	$\beta = 5$
Service execution mean value	7.5 sec
Service execution standard deviation	4.3 sec

6.3 Throughput vs. Concurrency Level

In this experiment we compare the overall throughput of the DSGT protocol and the conventional 2PL protocol under different concurrency levels. The overall throughput is measured by the number of terminated transactions per second. Figure 12 shows that DSGT_EdgeChasing and DSGT_PreScheduling in average have a much higher throughput than 2PL. The throughput of DSGT_EdgeChasing however, decreases dramatically when the conflict probability increases, as more and more transactions get delayed at commit time.

6.4. Communication Cost vs. Concurrency Level

The use of edge chasing algorithm for detecting global

waiting cycles imposes some overhead in terms of communication cost for propagating the tokens. Similarly, the use of pre-scheduling to avoid dependency cycles requires communication with the involved WS-Schedules to detect potential timing conflicts. In the experiment shown in Figure 13 we measure the communication cost in terms of average number of messages exchanged between coordinators and schedulers. The number of messages increases dramatically with the edge chasing approach as the dependency conflicts among transactions increase. The communication overhead of pre-scheduling is in average much less than the overhead of the edge chasing approach.

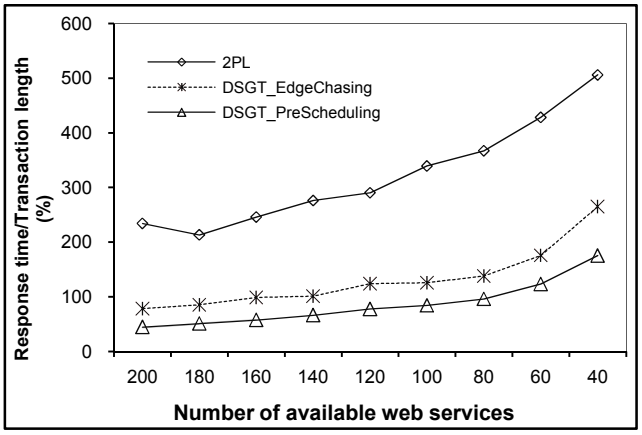

Figure 11. Response Time vs. Concurrency Level

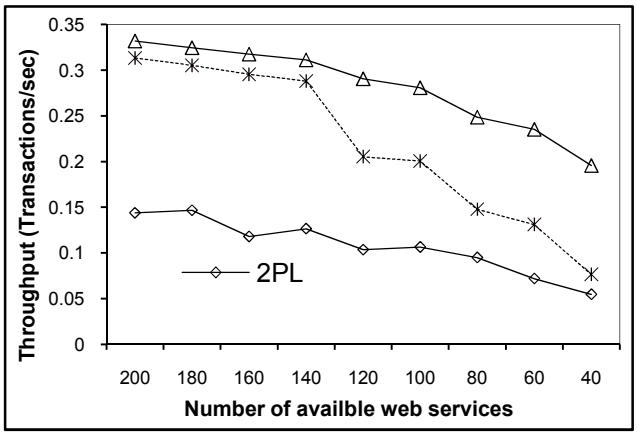

Figure 12. Throughput vs. Concurrency Level

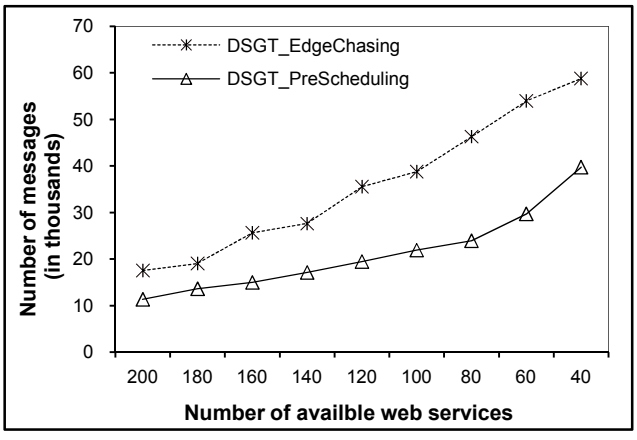

Figure 13. Communication Cost Comparison

6.5. Throughput vs. Transaction Complexity

In this experiment we study the impact of the transactions complexity (i.e. number of involved services) on the throughput of the applied concurrency method. Therefore, we repeated the experiment three times with the maximum number of composed services equals to: 10, 20 and 30 per transaction. The results shown in Figure 14 indicate that the throughput of all methods decreases when the number of involved services per transaction increases. However, in all cases, the throughput of the DSGT_EdgeChasing and DSGT_PreScheduling remains higher than the throughput of the 2PL protocol.

6.6. Throughput vs. Service Execution Duration

In this experiment we study the impact of the service execution duration on the throughput of the applied concurrency method. For this purpose, we repeat the experiment with the standard deviation of the service duration distribution equals to: 4.3, 7.6, 13, 17.3 seconds (see Figure 15). The results shown in Figure 16 indicate that the throughput of all methods decreases as the standard deviation increases. The throughput of the DSGT_EdgeChasing decreases much faster than the DSGT_PreScheduling. However, in all cases the throughput of the DSGT_PreScheduling remains higher than the throughput of the 2PL protocol in all cases.

6.7. Summary of the Results

The results of the experimental evaluation have shown that the distributed serialization testing graph protocol outperforms the conventional distributed 2PL protocol in terms of overall throughput and average response time. The results have also shown that using edge chasing approach for detecting and resolving global dependency cycles performs well with low level of conflicts. The performance of this approach decreases as the conflict level increases, while its communication overhead increases significantly. Therefore, this method for handling global dependency cycles is only useful in small environments like enterprise-enterprise business transactions where the probability that concurrent transactions get into transactional conflict is not very high. The pre-scheduling method performs better than the edge chasing method in all cases even with high levels of conflict. The pre-scheduling method, therefore, fits well to open and dynamic environments, where the level of conflicts is not predictable. This method is also useful for business applications with tight time constraints. A disadvantage of the pre-scheduling method lies in the complexity of its implementation, as it requires extending the WS-Coordinator's and WS-Scheduler's functionality to schedule service invocations in a timely manner.

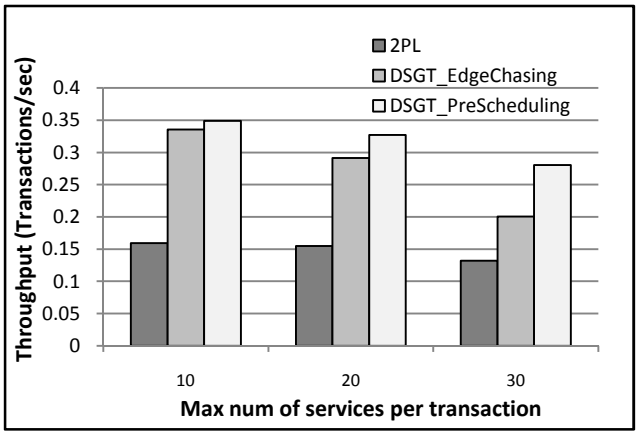

Figure 14. Throughput vs. Transaction Complexity

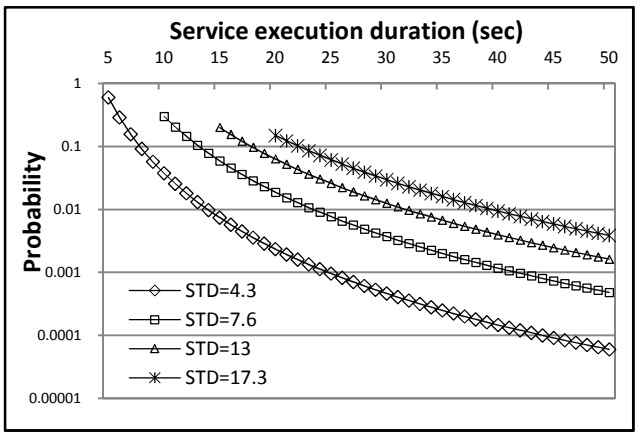

Figure 15. Distribution of Web Service Execution

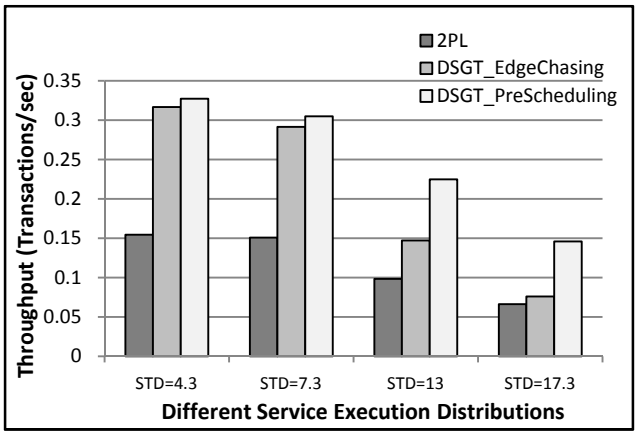

Figure 16. Throughput vs. Execution Distribution

7 Related Work

Ensuring a fault-tolerant execution of Web service transactions has been the focus of recent research work (e.g. [6], [14], [28] and [8]). The adopted transaction models in these works rely on the notion of compensations [10, 9, 7] which are triggered whenever a subset of a transaction fails. Compensations are introduced either at the client level as part of the business process execution [26] or on both, client and participant sides [27]. However, maintaining consistency of the concurrent transactions is neither addressed by these papers nor by the existing industrial specifications. An advanced database transaction model that deals with both atomicity and consistency of distributed applications is the ConTract model [29]. This model provides a user defined mechanism to control the correctness of the distributed transaction. The transaction initiator specifies so-called invariant predicates which have to remain unchanged from his application specific view to insure correctness. However, this model is not practical for loosely-coupled and autonomous Web services, where service providers cannot accept such predicates on their local resources by the clients. Moreover, as the service implementation is usually not visible to the service consumers, it is difficult to specify the right invariant predicates. In our solution, such predicates are specified by the service provider in the conflict matrix.

More recent work addressing the concurrency control problem of Web services is the work in [17] and [11]. Both solutions share the idea of handing over the concurrency control to the transaction coordinators, who in turn maintain and update local partial views of the global serialization graph by direct communication among them. The main disadvantage of these approaches is that they rely on information exchange among independent transactions to decide upon committing or aborting transactions. We argue that the assumption that independent transactions would like to exchange information about their own business relations and activities is unrealistic. The exchanged dependency information can be interpreted as mission-critical information such as confidential contracts between organizations. In contrast to this approach, our solution separates the roles of the transaction coordinators (commitment protocol) and transaction schedulers (concurrency control protocol) and does not require any direct communication or information exchange between independent transactions. Deciding upon committing or canceling transactions as well as detecting possible global dependency cycles are accomplished in our approach without disclosing any business related information.

8 Conclusion

Transactional dependencies can dynamically emerge between Web service-based business transactions due to the isolation relaxation property of advanced transaction models. It is important to take these dependencies into account to avoid inconsistency problems. In this paper we proposed efficient solutions to this problem. We extended the current Web service transaction framework to support concurrency control on service level. We also proposed fully distributed concurrency control protocol that can be deployed in the extended framework. Two methods for handling global dependency cycles in a fully distributed manner were presented. Empirical results of extensive simulations show that the proposed solutions perform well in different levels of dependency degrees.

REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. Web Services: Concepts, Architectures and Applications. Springer-Verlag, 2004.

[2] M. Alrifai, P. Dolog, and W. Nejdl. Transactions Concurrency Control in Web Service Environment. ECOWS, 2006.

[3] M. Alrifai, W.-T. Balke, P. Dolog, and W. Nejdl. Non-blocking Scheduling of Web Service Transactions. ECOWS, 2007.

[4] W.-T. Balke and M. Wagner. Cooperative Discovery for User-Centered Web Service Provisioning. ICWS, 2003.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[6] S. Bhiri, O. Perrin, and C. Godart. Ensuring Required Failure Atomicity of Composite Web Services. WWW, 2005.

[7] A. Elmagarmid. Database Transaction Models for Advanced Applications. San Mateo: Morgen Kaufmann, 1992.

[8] H. Erven, G. Hicker, C. Huemer, M. Zaptletal. The Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an Extension to the Web Services-BusinessActivity Specification. ICWS, 2007.

[9] G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. ICDE, 1996.

[10] J. Gray. The Transaction Concept: Virtues and Limitations. VLDB, 1981.

[11] K. Haller, H. Schuldt, and C. Türker. Decentralized Coordination of Transactional Processes in Peer to Peer environments. CIKM, 2005.

[12] E. Knapp. Deadlock Detection in Distributed Databases. ACM Computer Surveys 19, no. 4 (1987).

[13] F. Leymann and D. Roller. Production Workflow - Concepts and Techniques. Prentice Hall, 2000.

[14] F. Montagut and R. Molva. Augmenting Web Services Composition with Transactional Requirements. ICWS, 2006.

[15] E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Massachusetts Institute of Technology, 1981.

[16] M. Papazoglou. Web Services and Business Transactions. World Wide Web 6, no. 1 (2003): 49-91.

[17] S. Choi, H. Jang, H. Kim, J. Kim, S. Kim, J. Song, and Y. Lee. Maintaining Consistency under Isolations Relaxation of Web Services Transactions. WISE, 2005.

[18] G. Weikum. A theoretical foundation of multi-level concurrency control. PODS 1986.

[19] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Managment. ACM Transactions on Database Systems 16, no. 1 (1991).

[20] G. Weikum and G. Vossen. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. Morgen Kaufmann, 2001.

[21] G. Weikum and H.-J. Schek. Architectural Issues of Transaction Management in Multi-Layered Systems. VLDB, 1984.

[22] OASIS Web Service Atomic Transaction (WS-AtomicTransaction), 2007, http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os.pdf

[23] OASIS Web Service Business Activity (WS-BusinessActivity), 2007, http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec.pdf

[24] OASIS Web Service Coordination (WS-Coordination), 2007, http://docs.oasis-open.org/ws-tx/wscoor/2006/06

[25] OASIS Web services Business Process Execution Language, Version 2.0, 2007, http://docs.oasis-copen.org/wsbpel/2.0/wsbpel-v2.0.pdf

[26] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. P. Buchmann. Extending BPEL for Run Time Adaptability. EDOC, 2005.

[27] M. Schäfer, P. Dolog, and W. Nejdl. Engineering Compensations in Web Service Environment. ICWE, 2007.

[28] S. Bhiri, O. Perrin and C. Godart. Extending workflow patterns with transactional dependencies to define reliable composite Web services. iiWAS, 2006.

[29] H. Wächter and A. Reuter. The ConTract model. In Database Transaction Models For Advanced Applications, A. K. Elmagarmid, Ed. San Mateo: Morgen Kaufmann, 219-263, 1992.

Mohammad Alrifai received the B.Sc. degree in Computer Engineering from Baghdad University of Technology in 1998 and the M.Sc. degree in Computer Science from the University of Saarland, Germany in 2005. Since 2006 he has been a Ph.D candidate and researcher at the L3S Research Center in Hannover, Germany. His areas of research interest include service-oriented architectures and services computing with the focus on transactional management and Quality of Service aspects.

Peter Dolog has been associate professor of computer science at the Aalborg University since 2008, and assistant professor at the same university from 2006 to 2008. He received his doctoral degree in computer science from the University of Hannover (summa cum laude) in 2006 where he also worked as a researcher from 2002 to 2006. He studied computer science at the Slovak University of Technology in Bratislava. He worked as visiting researcher at DFKI Kaiserslautern, Politecnico di Milano, and Free University of Amsterdam. Peter Dolog heads the IWIS group (http://iwis.cs.aau.dk/) and does research in the areas of personalization and recommendation strategies on the web, web engineering, service oriented architecture, transactional middleware and algorithms for web services, and information systems. He has been program committee member and reviewer of numerous international conferences and journals.

Wolf-Tilo Balke currently is a full professor at Technische Universität Braunschweig and a director of the L3S Research Center, Hannover, Germany. Before that he was a research fellow at the University of California at Berkeley. His research is in the area of information systems and service provisioning, including preference-based database retrieval algorithms and ontology-based discovery and selection of Web services. Wolf-Tilo Balke is the recipient of two Emmy-Noether-Grants of the German Research Foundation (DFG) and the Scientific Award of the University Foundation Augsburg. He has received his B.A. and M.Sc. degree in mathematics and a PhD in computer science from University of Augsburg, Germany.

Wolfgang Nejdl (born 1960) has been full professor of computer science at the University of Hannover since 1995, and associate professor at the RWTH Aachen from 1992 to 1995. He studied computer science at the Technical University of Vienna, and worked as visiting researcher / professor at Xerox PARC, Stanford University, University of Illinois at Urbana-Champaign, EPFL Lausanne, and at PUC Rio. Prof. Nejdl heads the L3S Research Center (http://www.L3S.de/) and does research in the areas of search and information retrieval, information systems, semantic web and databases. He published more than 220 scientific articles, and has been program chair, program committee and editorial board member of numerous international conferences and journals.